Journal of Computational and Applied Mathematics 232 (2009) 638–654

Contents lists available at ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

Experimental study of scheduling with memory constraints using hybrid methods

J. Berlińska [a], M. Drozdowski [b,*], M. Lawenda [c]

[a] Faculty of Mathematics and Computer Science, Adam Mickiewicz University, Umultowska 87, 61-614 Poznań, Poland

[b] Institute of Computing Science, Poznań University of Technology, Piotrowo 2, 60-965 Poznań, Poland

[c] Poznań Supercomputing and Networking Center, Noskowskiego 10, 61-704 Poznań, Poland

ARTICLE INFO

Article history:
Received 3 December 2008

MSC:
68M14
90B35
90B20
90C27

Keywords:
Parallel processing
Scheduling
Divisible loads
Memory limitations
Multiple installments

ABSTRACT

In this paper we study divisible load scheduling in systems with limited memory. Divisible loads are parallel computations which can be divided into independent parts processed in parallel on remote computers, and the part sizes may be arbitrary. The distributed system is a heterogeneous single level tree. The total size of processor memories is too small to accommodate the whole load at any moment of time. Therefore, the load is distributed in many rounds. Memory reservations have block nature. The problem consists in distributing the load taking into account communication time, computation time, and limited memory buffers so that the whole processing finishes as early as possible. This problem is both combinatorial and algebraic in nature. Therefore, hybrid algorithms are given to solve it. Two algorithms are proposed to solve the combinatorial component. A branch-and-bound algorithm is nearly unusable due to its complexity. Then, a genetic algorithm is proposed with more tractable execution times. For a given solution of the combinatorial part we formulate the solution of the algebraic part as a linear programming problem. An extensive computational study is performed to analyze the impact of various system parameters on the quality of the solutions. From this we were able to infer on the nature of the scheduling problem.

1. Introduction

In this paper we study scheduling divisible loads in systems with limited memory sizes. Divisible load (DL) model represents parallel computations which can be divided into parts of arbitrary sizes, and the parts can be processed independently in parallel. These two simple assumptions on the nature of the parallel application have deep implications. Namely, the grains of parallelism are negligibly small because the load part sizes may be arbitrary. Since the parts can be processed independently, there are no data dependencies or other kinds of precedence constraints in the computation. Parallel processing of big volumes of data conforms with DL model. The processed data is generally called *load*. Divisible load model originated in the late 1980s [1,2]. In [1] DL model has been applied to represent distributed computations in a network of workstations. In publication [2] a chain of intelligent sensors was considered. In both cases the problem was how to partition the computations so that the whole processing time is as short as possible. On one hand distributing the computations reduces processing time by employing additional computers. On the other hand, distributing the computations takes time. Hence, the problem is what load quantities should be sent to which processors. The mathematical models proposed in the early publications were computationally tractable and boiled down to systems of linear equations.

* Corresponding author.
E-mail addresses: Joanna.Berlinska@amu.edu.pl (J. Berlińska), Maciej.Drozdowski@cs.put.poznan.pl (M. Drozdowski), Marcin.Lawenda@man.poznan.pl (M. Lawenda).

0377-0427/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.cam.2009.07.004

Later on, DL model has been applied to analyze performance of various computer network topologies, systems with parameters varying in time, limited memory sizes, to schedule computations in minimum monetary costs, and other. Overall, the divisible load theory (DLT) delivered a generic and versatile method of analyzing a broad class of parallel computations. Surveys of DLT can be found, e.g., in [3–5].

Scheduling divisible loads in systems with limited memory sizes was first considered in [6] where a heuristic called Incremental Balancing Strategy was proposed. It was assumed that all the processors always take part in the computation, the sequence of sending the load pieces to the processors is given, and that the whole load fits in the memory buffers of the computers. Furthermore, a simple linear model of communication delay was assumed. A more general affine communication time model including communication startup times was analyzed in [7]. A linear programming formulation was given which delivers optimum load partitioning under affine communication time model, and for a given sequence of load distributing. The problem of constructing optimum set of processors participating in the computation was shown to be **NP**-hard in [8, 14]. A branch-and-bound (BB) algorithm and a bunch of heuristics have been proposed and experimentally evaluated in [8] for the problem of single-round divisible load distributing with arbitrary communication sequence, and arbitrary set of participating processors. In this paper we assume that the whole load is too big to store it in the memories of the computers at the same moment. Therefore, it is distributed and processed in many small pieces, each of which fits in the computer memory. This organization of computations is called a *multi-round* or *multi-installment* processing [13,14]. Multi-round processing of divisible loads in systems with limited memory has been analyzed in [9]. An affine communication delay function was assumed, the set of processors taking part in the computation and the sequence of communicating with them was arbitrary. A branch-and-bound, and genetic (GA) algorithms have been proposed to solve this problem. However, memory management has been simplified in [9] to make the mathematical model tractable (we discuss it in more detail in the next section).

In this paper we study scheduling divisible loads in a heterogeneous star system (also called a single level tree) with limited memory buffers. We assume that the communication delay is an affine function of the amount of transferred load. The set of processors taking part in the computation, and the sequence of sending load chunks to them can be arbitrary and must be selected by the scheduling algorithm. Moreover, we assume that memory reservations and releases have realistic block nature (it is detailed in the next section). We propose two algorithms to solve our problem: an optimization branch-and-bound algorithm (BB) which guarantees optimality of the solution, and an approximate genetic search algorithm (GA). Though it can be proved that the former algorithm (BB) delivers optimum solutions, it is practically unusable due to its complexity. The latter algorithm (GA) delivers good quality solutions only on average, but it is more practical considering the execution time. We propose the GA not only to define yet another metaheuristic solving some combinatorial optimization problem, but also to gain insight into the features of near-optimum solutions, and hence, the nature of our scheduling problem. An extensive computational study was conducted to analyze practical features of the scheduling problem important for processing large divisible computations.

The rest of the paper is organized as follows. In Section 2 we formulate the problem formally. In Section 3 methods of solving the problem are presented and discussed. We report on the results of the computational experiments, and the insights into the nature of the problem in Section 4. In the last section we summarize earlier results and provide conclusions.

2. Problem formulation

In this work we assume that each processing element is equipped with a CPU, some memory, and hardware front-end for managing network communications (e.g. NIC and DMA). The CPU and network hardware can work in parallel so that simultaneous computation and communication is possible. The words processing element, processor, and computer will be used interchangeably. We assume star interconnection. In the center of the star a processor P_0 called originator is located. The originator is connected to a set $\{P_1, \ldots, P_m\}$ of processors. Initially the originator has some volume V of load to be processed. The load is sent directly from the originator to the processors. The star topology may represent a cluster of workstations connected via a local area network, a set of CPUs in an SMP system sharing a bus, a distributed computation with the originator as a master, and the computers as workers in a grid environment. We assume that only processors $P_1, \ldots, P_m$ perform computations, and the originator does no computing. Were it otherwise, the computing capability of the originator can be represented as an additional processor. For simplicity of mathematical models we assume that the time of returning the results to the originator is negligible. The processor and its communication link to the originator are characterized by the parameters: A_i – computing rate (inverse of speed, e.g., in seconds per byte), B_i – size of available memory (expressed, e.g., in bytes), C_i – communication rate (inverse of bandwidth), S_i – communication startup time (e.g. in seconds). The process of load distribution consists in sending pieces of the load to the processors for remote computation. Words installment, chunk, message, piece of load, communication will be used interchangeably. The transmission time of a load chunk of size α (e.g. bytes) sent to processor P_i is $S_i + \alpha C_i$. The same amount of load is computed on P_i in time αA_i.

Now let us analyze memory management. We assume that memory is allocated from the operating system pool at the beginning of the communication comprising the load chunk, and it is released to the operating system after the end of computation on the load chunk. The size of the load which arrived at a processor may not exceed the amount of available memory. The simplest approach to modeling memory usage is to assume that only one chunk is held by a processor at a time (cf. Fig. 1a). Hence, the chunk of size α_i may use all the available memory and a constraint $\alpha_i \leq B_i$ is imposed, for processor

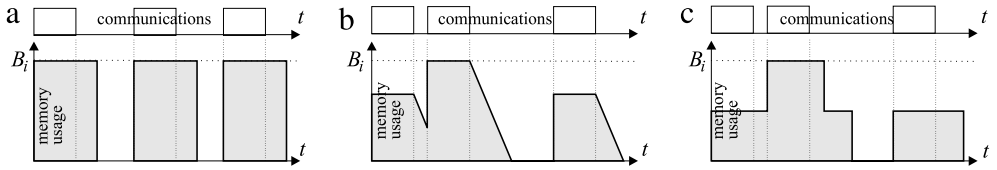

Fig. 1. Memory management: (a) each chunk uses whole buffer, (b) memory gradually released, (c) block memory releases.

Table 1
Summary of notation

m	Number of processors;
V	Volume of load;
A_i, B_i, C_i, S_i	Computing rate, memory size, communication rate, communication startup time of processor P_i, respectively;
$\mathcal{H}(i, t)$	The set of chunks received by P_i and not completed by time t;
σ	Communication sequence (constant in (1)–(9));
$\sigma(i)$	Index of the processor receiving the ith chunk in (1)–(9);
n	Length of communication sequence σ (constant in (1)–(9));
n_i	Number of chunks sent to processor P_i (constant in (1)–(9));
$\rho(i, k)$	Global number of the kth chunk received by processor P_i, i.e. mapping from local chunk numbers on P_i to the global numbers (constant in (1)–(9));
α_i	The ith chunk size (variable in (1)–(9));
T_{max}	Schedule length (variable in (1)–(9));
t_i	Time when sending of message i (global number) starts (variable in (1)–(9));
f_{ik}	Time when processing of message k (local number) on processor P_i finishes (variable in (1)–(9)).
δ_{ij}	Advancement of overlap interval introduced with chunk j on P_i (constant in formulation (1)–(9)).
z_{ij}	The last chunk overlapping with chunk j on P_i (constant in (1)–(9), derived from δ_{ij} in the algorithms).
$n_{\text{MIN}} = \lceil \frac{V}{\max_i\{B_i\}} \rceil$	Minimum number of load chunks
$m' \leq m$	The number of different used processors

P_i. This approach was used in [8,6]. Yet, it is insufficient in multi-installment processing when many messages may arrive at the processor over time. The load can be gradually uploaded, new and old buffers can be swapped without stopping the computations. Consequently, the load chunk sizes may interact with each other. In [9] it was assumed that the received load *together* shall not exceed memory size B_i. However, to make the problem easier it was also assumed that memory is released to the operating system with very fine granularity equal to the load unit. Consequently, memory occupation was decreasing linearly during computations (cf. Fig. 1b). With such a simplification optimum load chunk sizes could be calculated using linear programming for a given communication sequence. Still, this way of releasing memory is rather unusual because a memory block obtained from the operating system is indivisible, and cannot be returned in pieces of fine size. In this paper we assume that memory allocation and release have block nature (cf. Fig. 1c). When a chunk of size α_j is about to arrive to a processor, a block of α_j load units is requested from the operating system. This block exists in the memory pool of the application until finishing computation on chunk j. On completion of chunk j a block of size α_j is released to the operating system. The sizes of coexisting memory blocks cannot exceed limit B_i. In other words, for each moment t, $\sum_{l\in\mathcal{H}(i,t)} \alpha_l \leq B_i$, where $\mathcal{H}(i, t)$ is the set of chunks received by P_i and not completed by time t. We will be saying that chunks simultaneously existing in memory *overlap*. Thus, $\mathcal{H}(i, t)$ is the set of chunks overlapping at time t on P_i.

In a very preliminary version [10] of this study, our scheduling problem has been formulated as a mixed nonlinear mathematic program. Here we will formulate this problem in a simpler way by exploiting the fact that a solution of our problem consists of two parts: combinatorial and algebraic.

Let us introduce necessary assumptions and notation. The load is delivered to the processors in a sequence of communications. The sequence may be arbitrary, which means that some processors may be excluded from the computations, while some other processors may receive the load more often than the others. If the message is received by a processor without any load in the buffer, then computations start immediately after the end of communication. If the buffer already stores some unprocessed chunks, then the processor switches from computing one load chunk to the next one without idle time in the computations. Idle times may arise between the communications when processor memory occupation is maximum, and no new load may be uploaded to any processor. We assume that the sequence of processing the chunks on a given processor is the same as the sequence in which they were received. Let us assume that the sequence $\sigma = (\sigma(1), \ldots, \sigma(n))$ of the communications to the processors is given, where $\sigma(i)$ is the index of the processor receiving the ith chunk. The numbers of the load chunks as they are sent off the originator will be called *global* numbers. For simplicity of notation also *local* numbering of the chunks received by a certain processor will be used. Function $\rho(i, j)$ is a mapping from processor P_i local chunk number j to the global numbering. Let us assume that z_{ij} denotes the last chunk which is overlapping chunk j (local number) on processor P_i. Consequently, $j \leq z_{ij} \leq n_i$ for $i = 1, \ldots, m$, $j = 1, \ldots, n_i - 1$. In Table 1 we summarize the notation introduced so far and used in the rest of the paper. Our problem can be formulated in the following way.

minimize $T_{\max}$
subject to

$$t_1 = 0 \tag{1}$$

$$t_i \geq t_{i-1} + S_{\sigma(i-1)} + C_{\sigma(i-1)}\alpha_{i-1} \quad i = 2, \ldots, n, \tag{2}$$

$$f_{ik} \geq t_{\rho(i,k)} + S_i + C_i\alpha_{\rho(i,k)} + A_i\alpha_{\rho(i,k)} \quad i = 1, \ldots, m, \quad k = 1, \ldots, n_i, \tag{3}$$

$$f_{ik} \geq f_{i,k-1} + A_i\alpha_{\rho(i,k)} \quad i = 1, \ldots, m, \quad k = 2, \ldots, n_i, \tag{4}$$

$$f_{ij} \geq t_{\rho(i,z_{ij})} \quad i = 1, \ldots, m,\ j = 1, \ldots, n_i - 1 \tag{5}$$

$$f_{ij} < t_{\rho(i,z_{ij}+1)} \quad i = 1, \ldots, m,\ j = 1, \ldots, n_i - 1 \tag{6}$$

$$\sum_{k=j}^{z_{ij}} \alpha_{\rho(i,k)} \leq B_i \quad i = 1, \ldots, m,\ j = 1, \ldots, n_i \tag{7}$$

$$V = \sum_{i=1}^{n} \alpha_i \tag{8}$$

$$T_{\max} \geq f_{in_i} \quad i = 1, \ldots, m. \tag{9}$$

Variables α_i in the above formulation define load partitioning resulting in minimum schedule length for the communication sequence σ. Inequalities (1) and (2) determine the moment when sending of the ith chunk starts. Constraints (3) and (4) determine the earliest time moment f_{ik} when computation on chunk k of P_i finishes. Inequality (5) guarantees that processing of chunk j is not finished before starting message z_{ij}, and it is finished before chunk $z_{ij} + 1$ starts arriving by inequality (6). By inequalities (7) memory limits are observed. No load remains unprocessed by (8). Schedule length is not shorter than the completion time on any processor by constraints (9). Formulation (1)–(9) is a linear program for the given communication sequence σ, and the overlap information z_{ij}. Calculating the optimum values of α_i, t_i, f_{ik}, $T_{\max}$ from (1)–(9) is the algebraic part of our problem. Obtaining the two elements of communication sequence σ and z_{ij} constitutes combinatorial part of our problem. This is in sharp contrast with the complexity of memory management models used in [9,8] for which linear programs were needed to calculate load partition for a given communication sequence σ. Thus, representation of block memory management, and chunk overlap made the mathematical model much more involved. Note that (1)–(9) is very general and may cover various scenarios of optimum memory management. For example, it is capable of representing a number of independent buffers of equal or different sizes swapped on the processors. Splitting the problem into combinatorial part and algebraic part which can be solved by linear programming is a foundation of the solution methods proposed in the next section.

3. Algorithms

In the previous section we established that for a given communication sequence σ, and given values of variables z_{ij} encoding chunk overlap, optimum chunk sizes α_i can be calculated by an LP. Hence, our problem can be solved by a tandem of methods. The first solves the combinatorial part of the problem by selecting activation sequence σ, and chunk overlap z_{ij}. The second part solves the LP for the given σ and z_{ij}.

In the actual algorithms solving our problem a more convenient encoding of the *depth of overlap* was used. Note that if chunk j is overlapping with z_{ij} then due to preserving the order of computing the chunks, it is also overlapping with all the intermediate chunks received between j and z_{ij} on processor P_i. Moreover, if z_{ij} is the last chunk overlapping j, then the latter chunks cannot overlap with j anymore. The sequences of overlapping chunks create compact intervals. Instead of recording the index z_{ij} of the last chunk overlapping with chunk j, a differential encoding was used. Integer variables $\delta_{ij} \geq 0$ denote by how many chunks the overlapping front is forwarded with chunk j on P_i. For the given values of δ_{ij}, the index of the last overlapping chunk is $z_{ij} = \min\{n_i, \max\{z_{i,j-1} + \delta_{ij}, j + \delta_{ij}\}\}$, and $z_{i0} = 1$. This new differential overlap encoding is used in both methods presented in the following sections. For given σ and δ_{ij}, distribution of the load can be obtained from the linear program (1)–(9).

3.1. Branch-and-bound algorithm

A branch-and-bound algorithm (BB) is a standard technique applied in solving combinatorial optimization problems. In BB algorithm a branching rule divides the set of possible solutions until distinguishing unique solutions. The pruning (or bounding) rule eliminates sets of solutions which are certainly not better than some already known solution, or are infeasible.

In our problem one has to determine a sequence of communications and chunk overlapping. Communication sequences were built by appending a new processor to some already constructed leading sequence. For example, sequence $\sigma = (P_a, \ldots, P_z)$ represents all the solutions beginning with communication sequence σ. This set is partitioned by appending a

communication to any processor from set $\{P_1, \ldots, P_m\}$. Thus, the set of solutions represented by σ is branched into subsets of solutions beginning with sequences: $(P_a, \ldots, P_z, P_1), \ldots, (P_a, \ldots, P_z, P_m)$. For each communication sequence chunk overlapping on the used processors must be decided. All overlaps possible in the new encoding were enumerated in the following way. For processor P_i overlap is a vector $(\delta_{i1}, \ldots, \delta_{in_i})$. A sequence $(\delta_{i1}, \ldots, \delta_{ij})$ encoding the overlap for the first j chunks received by P_i, was branched into overlap encoding strings $(\delta_{i1}, \ldots, \delta_{ij}, 0), \ldots, (\delta_{i1}, \ldots, \delta_{ij}, n_i - 1 - \max\{j, z_{i,j-1}\})$. Thus, the BB algorithm uses a double branching scheme: for communication sequences, and for the chunk overlaps.

Enumeration of possible solutions was pruned by two methods. For a given sequence σ a lower bound $LB(\sigma)$ on the schedule length was calculated. The startup times in σ were summed up: $\tau_1 = \sum_{i=1}^{n} S_{\sigma(i)}$. The maximum load V_1 that could be processed during the communication startup times is $V_1 = \sum_{i \in \sigma}(\tau_1 - \sum_{j=1}^{g(i)} S_{\sigma(j)})/A_i$, where $g(i)$ is the index of the first communication to processor P_i in σ. The load must be sent from the originator in time at least $\tau_2 = V \min_{i=1}^{m}\{C_i\}$. In parallel with this communication, at most $V_2 = \frac{\tau_2}{\sum_{i=1}^{m} \frac{1}{A_i}}$ units of load could be processed. If $V - V_1 - V_2 = V_3 > 0$, then this remaining load V_3 will be processed in time at least $\tau_3 = \frac{V_3}{\sum_{i=1}^{m} \frac{1}{A_i}}$. The lower bound is equal to $LB(\sigma) = \tau_1 + \tau_2 + \max\{0, \tau_3\}$. Let T be the length of some already known solution. If $T \leq LB(\sigma)$ then successors of σ were discarded. Another mechanism used in sequence elimination was based on the maximum memory size $MEM(\sigma) = \sum_{i=1}^{n} B_{\sigma(i)}$ which could possibly become available in σ. If $MEM(\sigma) < V$, then it means that memory available for holding the load is insufficient, communication sequence σ is too short and must be expanded. In such a case the enumeration of the various overlap values was not attempted for the given σ. Observe that there are $O(m^n)$ communication sequences of length n for m processors. For each processor the number of possible ways of overlapping the communication chunks is also exponential in n_i. Due to the high computational complexity an upper bound n_{MAX} on length n of generated sequences was imposed. Note that this was done to make the BB algorithm more usable, and it was not needed to properly define the algorithm. Due to imposing the n_{MAX} limit not in all cases was BB able to deliver an optimum solution.

3.2. Genetic algorithm

The genetic algorithm (GA) is also one of the standard techniques used in solving combinatorial optimization problems. GA is a randomized algorithm using a set of operators transforming a population of solutions in the direction of improving quality. GA is defined by the way of encoding the solution, the set of genetic operators, stopping criteria, and several implementation-dependent tunable parameters.

In our implementation of GA solutions are encoded as pairs of strings. The first string is a sequence of processor indices encoding communication sequence σ. The second string O is encoding overlap of chunks. More precisely, $O(i)$ is the value of $\delta_{\sigma(i)j}$, where j is the number of load chunks sent to processor $P_{\sigma(i)}$ up to the ith chunk sent off the originator. The lengths of σ, O are equal, and can be adjusted by GA to construct the best solution. Fitness of the solution (called a chromosome) is measured as the inverse of schedule length $T_{\max}$ which is obtained from the linear program (1)–(9) for the given σ, O.

The genetic operators of GA applied here are selection, crossover, and mutation. The selection of the chromosomes for the new population is done by a combination of elitist and roulette wheel method and is strongly connected with the crossover operation. Chromosome j is selected for the crossover with probability $\frac{1}{T^j_{\max}} / \sum_{j=1}^{G} \frac{1}{T^j_{\max}}$, where $T^j_{\max}$ denotes the schedule length for chromosome j, and G is the size of the population. The total number of selected parents is Gp_C, where p_C is a tunable algorithm parameter called crossover probability. In the crossover operation the selected parents are randomly paired and combined. For example, let $[(\sigma_1(1), \ldots, \sigma_1(n')), (O_1(1), \ldots, O_1(n'))]$ and $[(\sigma_2(1), \ldots, \sigma_2(n'')), (O_2(1), \ldots, O_2(n''))]$ be two parent solutions, with communication sequence lengths n', n'', respectively. Let $k \leq n'$, $l \leq n''$ be two randomly chosen crossover points. The two offspring solutions are encoded by

$$[(\sigma_1(1), \ldots, \sigma_1(k), \sigma_2(l+1), \ldots, \sigma_2(n'')), (O_1(1), \ldots, O_1(k), O_2(l+1), \ldots, O_2(n''))],$$

and

$$[(\sigma_2(1), \ldots, \sigma_2(l), \sigma_1(k+1), \ldots, \sigma_1(n')), (O_2(1), \ldots, O_2(l), O_1(k+1), \ldots, O_1(n'))].$$

Note that because of choosing two crossover points l, k the offspring string lengths may be different than in their parents. The rest of the new population is selected by elitist method so that the best $(1 - p_C)G$ chromosomes are always preserved. The elitist component in the selection was necessary because very often the difference in solution fitness is small, and the best solutions may be lost in the randomized selection based on the schedule length only.

Mutation changes $E(t)p_M$ random genes (i.e. pairs $(\sigma(i), O(i))$) in the population to different values. Here $E(t) = \sum_{j=1}^{G} n^j(t)$ is the total number of genes in the population in generation t, $n^j(t)$ is the length of chromosome j in iteration t, and p_M is a tunable algorithm parameter called mutation probability.

The algorithm stops after a fixed number of iterations it_1. There is also a limit it_2 on number of iterations without an improvement in the quality of the best solution found so far. If iteration limit it_2 is reached before it_1, then the population is replaced with randomly generated chromosomes and the search is started from scratch (the best solution found so far is recorded).

GA is a randomized algorithm whose parameters must be tuned. We applied the following procedure. A set of 200 random instances with $m = 3, \ldots, 6$, $V = 20$, B_i uniformly distributed in [0, 10], A_i, C_i, S_i uniformly distributed in [0, 1], were

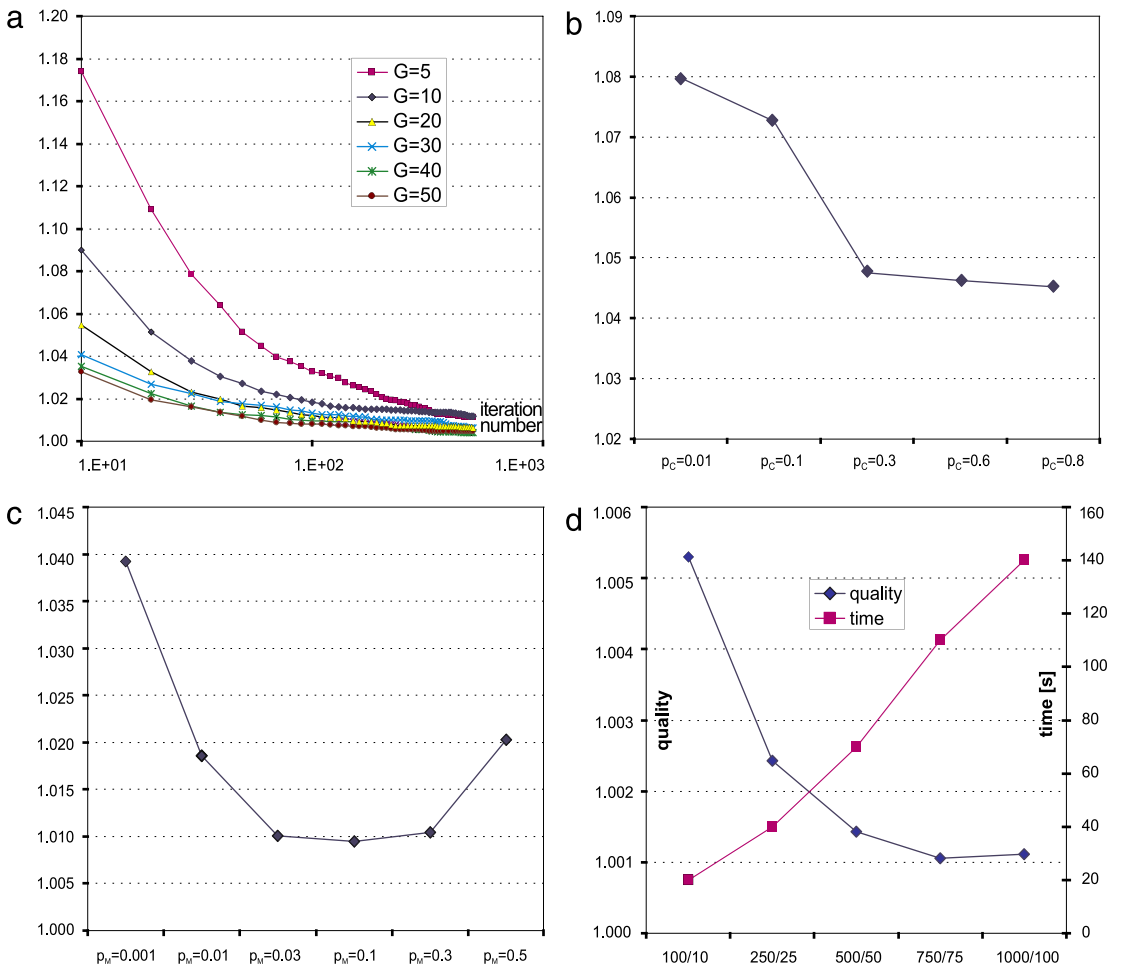

Fig. 2. GA tuning. (a) Solution quality vs. population size G, (b) solution quality at the 100th iteration vs. p_C, (c) solution quality at the 100th iteration vs. p_M, (d) solution quality and execution time for various iteration limits it_1/it_2.

generated and solved to the optimality by BB. The average relative distance of the schedule length $T_{\max}$ from the optimum length was the measure of the tuning quality. The tunable parameters were selected one by one. The process of selecting the tunable parameters is illustrated in Fig. 2. Intuitively, a big population size G should allow for finding good solutions in small number of iterations. On the other hand, maintaining big populations is computationally expensive. The population size $G = 20$ was selected as a compromise between the speed of convergence and the computational complexity (cf. Fig. 2a). To select the crossover probability, mutation operator was switched off. Crossover probability $p_C = 0.8$ was selected (Fig. 2b). Thus, it can be concluded that a majority of the population (80%) are offspring, and crossover is an effective optimization operator. After fixing G and p_C, mutation probability $p_M = 0.1$ was chosen (Fig. 2c). In Fig. 2d quality of tuning and execution time for various combinations of maximum number of iterations, and iterations without quality improvements are shown. Note that improving the average solution quality by 0.4% results in nearly 6-fold increase of the execution time. Hence, $it_1 = 100$, $it_2 = 10$ were selected as a compromise between quality and complexity.

3.3. BB and GA comparison

In this section we discuss advantages and limitations of BB and GA algorithms. BB guarantees optimum solutions, however, at considerable computational cost. In Fig. 3 we compare average execution time of BB and GA. In the case of BB execution time is shown as function of n_{MAX} (Fig. 3a). We use n_{MAX} because it turned out that the size of the search tree in BB is determined mainly by the limit n_{MAX}. As it can be seen even average execution time of BB for $n_{MAX} = 7$, $m = 8$ is of order of one day on a Pentium IV 1 GHz CPU. Hence, BB is not acceptable as a tool for studying features of great numbers of even moderate size instances. For the GA, execution time is shown vs. the length of the best obtained communication sequence. In Fig. 3b execution time vs. the number of processors m is shown.

From the tuning process described in the previous section we conclude that GA is capable of delivering high quality solutions on average. As it can be seen in Fig. 3 the running time of GA is much shorter than for BB. A disadvantage of

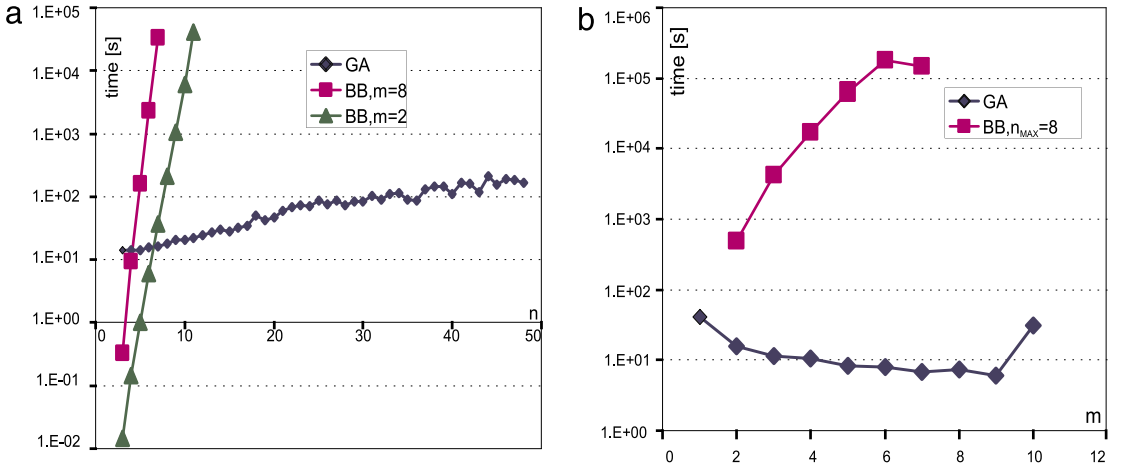

Fig. 3. GA and BB execution times (a) vs. sequence length, (b) vs. processor number m.

GA as an analysis tool is that it is a randomized algorithm. In the limit of infinite iteration number, all feasible solutions are reachable in a process of random solution transformations. The randomness in GA has the following disadvantages. Solutions which are easy to find by GA may be far from optimal. On the other hand, solutions which have complex structure may be too improbable to be built in finite iteration number. For example, the communication sequence may include some processor which is not present in the optimum solution because the probability of choosing any processor to the sequence is relatively high. Conversely, it is very unlikely that GA builds a long repetitive pattern of communications in the communication sequence because the probability of generating the sequence decreases exponentially with the sequence length. Moreover, for the same instance GA may return different solutions in consecutive runs. For example, for a set of 45 random instances each solved 20 times, the quotient $\frac{T_{\max}}{\overline{T_{\max}}}$ where $\overline{T_{\max}}$ is an average schedule length in all runs for a single instance, had the coefficient of variation 6%. Hence there is a dispersion of solutions on the order of several percent.

We finish this section with a conclusion that BB is nearly unusable even on very moderate size instances. GA has much shorter execution time, and in the range in which it could be compared against BB, the quality of the GA solutions is very good. Though GA has its limitations it is the only tool at hand capable of solving bulk numbers of instances in reasonable time. Therefore, we will use GA as a replacement of BB in the analysis of the scheduling problem features.

4. Analysis of the problem

In this section we present results of the computational study on the characteristic of the near-optimum solutions of our scheduling problem. We mainly concentrate on the features in the combinatorial part of the solutions: the communication sequence σ, and the vector of overlaps O. The studied features include:

- the need and the extent of the load chunk overlap,
- the length of communication sequence,
- the number of used processors,
- the set of used processors,
- the chunk sizes,
- instances parameters which make the problem easy, or hard, to solve.

We will draw conclusions both analytically and on the basis of experimental results. Both GA and BB use `lp_solve` linear programming package [11]. Over 30 000 instances were solved in the experiments on clusters of 15-75 PCs with Linux. Unless stated otherwise, the test data were generated in the following way. In the experiments involving analysis of the influence of system parameters A_i, B_i, C_i, S_i on solution characteristic, parameters A_i, B_i, C_i, S_i, were generated from $U(0, 1]$, i.e. uniform distribution within range (0,1]. The number of processors was generated from $U[1, 10]$, and all experiments were repeated for $V \in \{2, 5, 10, 20, 50\}$. In the experiments concerning certain parameter (say A) the parameter was fixed to a given value on all processors (e.g. $\forall i$, $A_i = 0.01$), and the remaining parameters were randomly generated as described above. For each combination of V, and value of the examined parameter (e.g. $A_i = 0.01$) 100 instances were generated. In the following sections we discuss features of the solutions.

4.1. Depth of overlap

By the depth of overlap we mean the number of the load chunks which interfere with each other. The depth of overlap is expressed by the values of δ_{ij} which can be converted to the values of the span of overlap $z_{ij} - j + 1$ for each chunk j on processor P_i (see Section 3). The existence of the overlap means that load must accumulate on the processors. It is of practical importance to verify if the accumulation of the load is actually necessary, and to what degree.

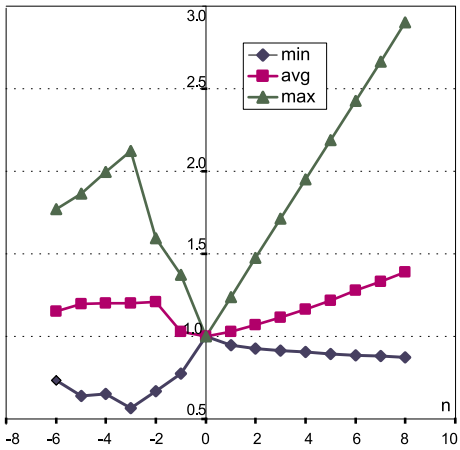

Fig. 4. $m = 1$, quality of the solutions with various communication sequence lengths, the best overlap, relative to coupled solutions.

4.1.1. Single processor considerations

Here we analyze the case of one computing processor ($m = 1$). Despite simplicity, this problem is not trivial because to construct a schedule one has to decide on the overlap of the load chunks, and their sizes.

Let us start with several simple observations which can help in reasoning about our problem. For simplicity of presentation we drop the subscripts related to processor indices. We denote processor parameters as A, B, C, S, and the overlap of chunk j by δ_j. If $\forall j,\ \delta_j = i$, then we will say that a solution has overlap i. We will be saying that solutions for which chunks overlap by not more than 1, i.e. $\forall j,\ \delta_j \leq 1$, have overlap at most 1.

Observation 1. *For $m = 1$ the schedule with the $n_{\mathrm{MIN}} = \lceil \frac{V}{B} \rceil$, and hence overlap 0, can be at most twice as long as the optimum schedule.* [12]

Observation 2. *For $m = 1$ there is no need for overlap greater than 1* [12].

A schedule with overlap 1 has $\forall j,\ \delta_j = 1$, and the chunks overlap with their direct predecessor and direct successor (if any). If chunk 1 has size α_1, then by (7) chunk 2 has size at most $\alpha_2 \leq B - \alpha_1$, chunk 3 has size at most $\alpha_3 \leq B - \alpha_2 = \alpha_1$, etc. Thus, if chunks have all their maximum sizes, then the size of all chunks is in fact determined by a single variable α_1. The size of processed load is $\frac{n}{2}B$ if communication sequence has even number n of messages, or it is $\frac{n}{2}B + \alpha_1$ if n is odd. Chunk sizes are coupled if $n = \lceil \frac{2V}{B} \rceil$, and the overlap is 1. Thus, the number of chunks is chosen minimum for the given overlap. We will say that solutions for $m = 1$ with $n = \lceil \frac{2V}{B} \rceil$, and overlap 1 are *coupled*.

Observation 3. *Schedule length for coupled solutions is at most 4 time worse than the optimum* [12].

The above observation gives an indication on the quality of schedules with $n = \lceil \frac{2V}{B} \rceil$ and overlap 1 in the worst case. In Fig. 4 quality of schedules for $m = 1$, various sequence lengths, and the best overlap chosen by BB algorithm is shown. The coupled solution quality is used as a reference, and is represented by the points at the coordinates (0, 1). For example, shorter sequences are shown on the negative part of horizontal axis. Solutions which are better than the coupled solutions are below 1 on the vertical axis. The best, the worst, and an average relative distance from the coupled solution is shown. The results in Fig. 4 represent 888 randomly generated instances with $A, C, S \sim U[0, 1]$, $B \sim U(0, 10)$, $V = 10$. As it can be seen, typically the best solutions are not very much better than the coupled ones. Increasing n beyond $\lceil \frac{2V}{B} \rceil$ is not reducing schedule length more than by approx. 13%. Thus, on average coupled solutions provide a simple and efficient method of solving the combinatorial part of our problem on $m = 1$ processor. Let us observe that optimum communication sequence length n can be smaller or greater than $\lceil \frac{2V}{B} \rceil$ depending on the instance.

4.1.2. Overlap on multiprocessors

In the preceding section it has been observed that no overlap greater than 1 is necessary if $m = 1$. However, for $m > 1$ arbitrarily big overlap may be necessary in optimum solutions.

Observation 4. *There are optimum solutions with arbitrarily deep overlap* [12].

Results collected from computational study performed with the use of GA on 19953 randomly generated instances with $A, B, C, S \sim U[0, 1]$, $m \sim U[1, 10]$ and $V \in \{2, 5, 10, 20, 50\}$ indicate, however, that the depth of the overlap is not very big. Table 2 lists the depth of the overlap of all chunks in all sequences of the solutions generated by GA for the above

Table 2
Relative frequency of the overlaps in all the chunks.

Overlap δ_{ij}	0	1	2	>2
Frequency	0.835	0.154	0.010	0.001

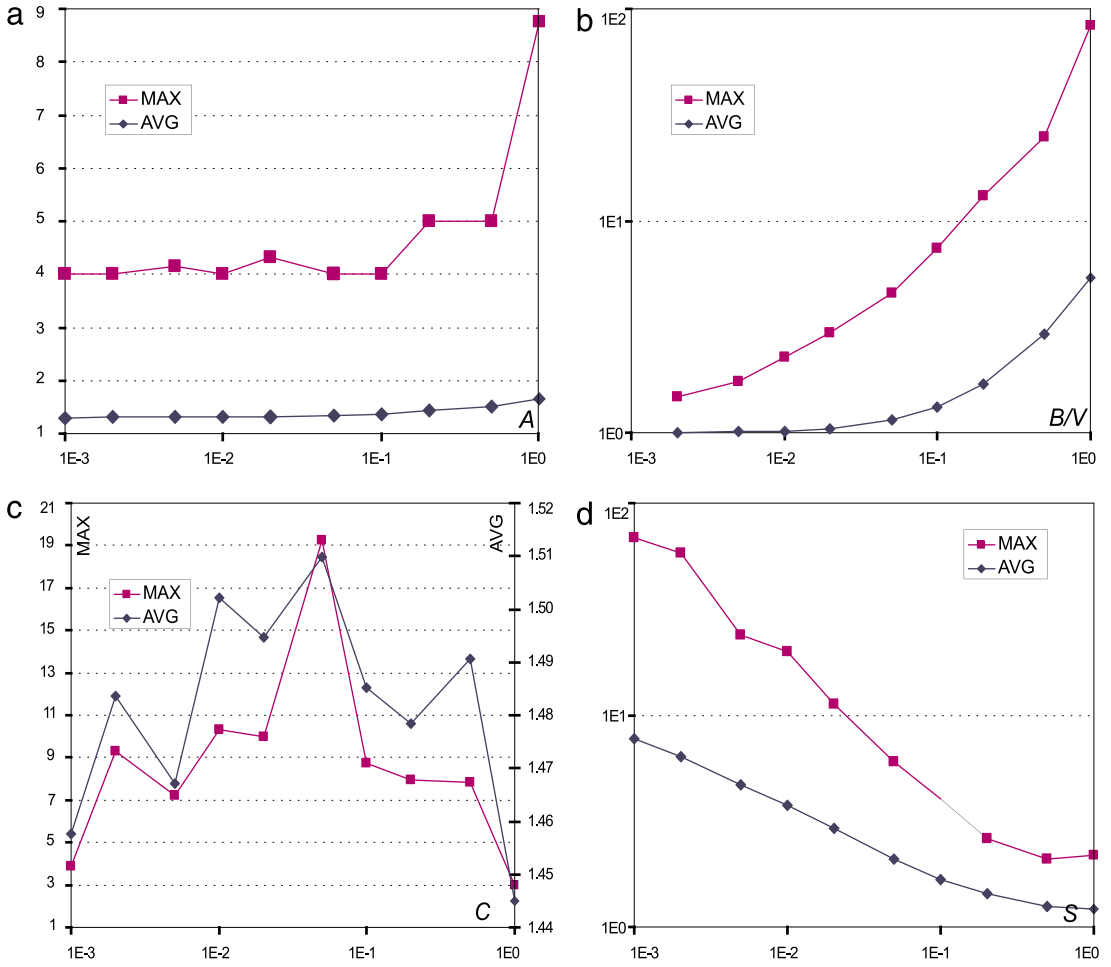

Fig. 5. Relative sequence length $\frac{n}{n_{\text{MIN}}}$ in the solutions of GA (a) vs. A, (b) vs. $\frac{B}{V}$, (c) vs. C, (d) vs. S.

instances. For the above experimental results we can conclude that overlap deeper than 1 is rare, because it constitutes approx. 1% of all chunks in all solutions.

The analysis of the depth of the overlap leads to the following conclusions: On a single processor the overlap is in {0, 1} [12], the solutions with overlap 0 only, or 1 only, cannot be arbitrarily bad, and solutions with $n = \lceil \frac{2V}{B} \rceil$, and $\forall j$, $\delta_j = 1$ are good on average. For multiple processors, the overlap may be arbitrarily deep in the worst case, but overlaps greater than 1 are rare in practice.

4.2. Length of the communication sequence

One of the important characteristics of the solution is the number of communications n. The length of the communication sequence depends on V, and B_is. Therefore, it seems reasonable to use some reference number of communications. Let us assume that the reference number of communication is n_{MIN}. We start with an observation.

Observation 5. *A communication sequence with minimum number of chunks* $n_{\text{MIN}} = \lceil \frac{V}{\max_i\{B_i\}} \rceil$ *can be arbitrarily bad for schedule length. The length n of the optimum communication sequence can be arbitrarily big in relation to* n_{MIN} [12].

Let us now analyze length n of communication sequences generated by GA. The values of relative communication sequence lengths $\frac{n}{n_{\text{MIN}}}$ are shown in Fig. 5. In Fig. 5a the average (AVG), and the longest (MAX) observed communication lengths are

shown for various A values. It can be seen that typically $\frac{n}{n_{\mathrm{MIN}}}$ is not very big. On average $n \approx 1.39 n_{\mathrm{MIN}}$, which is calculated over all instances of changing A. The length of the sequence grows with A, which is especially evident for the biggest registered relative lengths (MAX). This phenomenon can be attributed to the way of calculating n_{MIN}. For example, for $V = 1$, and $B_i \in (0, 1]$ the expected n_{MIN} is 2, and in extreme cases it can be just $n_{\mathrm{MIN}} = 1$. Thus, it is not a rare case that n_{MIN} is quite small. On the other hand, as processors get slower (A is increasing) it gets more and more profitable to use all m available processors. Thus, $\frac{n}{n_{\mathrm{MIN}}}$ grows with A. This increase is stronger for small V, and weaker for bigger V.

In Fig. 5b similar dependence is shown for changing $\frac{B}{V}$. The length of the communication sequence quickly increases with $\frac{B}{V}$. It is because, on one hand for $\frac{B}{V}$ approaching 1, n_{MIN} is also approaching 1, but as pointed out in Observation 5 other parameters of the system make it profitable to build sequences with $n \gg 1$. On the other hand, as $\frac{B}{V}$ approaches 0, more and more short communications must be made to send the load off the originator. Each message carries cost of startup S. Therefore, communication costs and startup in particular, dominate in the schedule length. To minimize this cost it is advantageous to send as few messages as possible. Hence, n tends to n_{MIN} when $\frac{B}{V}$ is decreasing. Similar observations can be made for big values of S (cf. Fig. 5d). For big S it is profitable to send as few messages as possible. This, in turn, exposes the need for big communication buffers. The behavior of $\frac{n}{n_{\mathrm{MIN}}}$ for small S must be contrasted with Fig. 5a. When $S \approx \frac{1}{2}$, as it is on average in Fig. 5a, then $\frac{n}{n_{\mathrm{MIN}}} \approx 1.39$. If $S = 0.001$, as in Fig. 5d, then $\frac{n}{n_{\mathrm{MIN}}} \approx 8$. This means that big startup time is a considerable disincentive to building long communication sequences.

In Fig. 5c dependence of $\frac{n}{n_{\mathrm{MIN}}}$ on C is shown. Note that this figure has two vertical axes. The shapes of *MAX*, and *AVG* are similar, but for the average case the changes are in the range of approximately 5%. This should be surprising because multi-installment divisible load processing was introduced to reduce the time of initial waiting for load, and consequently reduce influence of parameter C on performance of the distributed system. Growing value of C should be an incentive to build shorter messages and longer communication sequences. This tendency can be seen only for small values of C. Yet, in our setting of the experiments expected value of the startup times is $\approx \frac{1}{2}$ which is a disincentive to build long communication sequences as explained on the example of Fig. 5a, and Fig. 5d. Hence, the dependence of average $\frac{n}{n_{\mathrm{MIN}}}$ on C is very weak. Moreover, with growing C the algorithm tends to compensate increasing communication costs by sending fewer messages. Thus, initial waiting for the load is meaningless compared to the whole communication cost.

From the above analysis of the communication sequence length we draw the following conclusions: Startup times S_i are important element of communication time, and they constitute main disincentive to build long communication sequences. For startup times of the same order as communication time per unit of load (C), or computation time per unit of load (A) communication sequences have lengths $\approx 1.4 n_{\mathrm{MIN}}$. For small S the sequences can be approximately 8–10 times longer on average than n_{MIN}. Moreover, S_i, and B_i are in a sense coupled in determining system performance: Small B_is expose costs of communications including startups, big S_is expose the need for processors with big communication buffers.

4.3. Number of used processors

In this section we study the number m' of different processors used. This feature of a solution has a practical meaning. Considering big pools of processors available in contemporary grid and cluster systems it is of practical importance to know how many processors should be used, and how to adjust their number to changing characteristic of the system and application. It is not difficult to coin instances where only one processor may be used (e.g. because all other processors have their startup times greater than schedule length) or all processors must be used. It is a known fact from divisible load theory that if $\forall i, S_i = 0$, then computations can be started on arbitrary number of processors. On the other hand if $\exists i, S_i > 0$, then for single-installment processing using all processors is a matter of sufficiently big volume of load V. Thus, it may be intuitively expected that the number of used different processors m' should grow with decreasing startups and increasing V.

Let us now analyze the features of GA solutions. Relations between the ratio $\frac{m'}{m}$ and selected parameters are shown in Fig. 6. It can be seen in Fig. 6a that with growing amount of load V the number of different used processors is growing as could be intuitively expected. This result was confirmed in all experiments we performed. This has a practical consequence, that for bigger problems it is profitable to use more processors instead of relying on bigger number of load chunks n only.

The dependence of m' on A is show in Fig. 6b. Only for small problem sizes (small V) does m' increase with A. For small V only a few chunks need to be sent. Therefore, for small A the algorithm minimizes schedule length by selecting only a few processors with big memory buffers and fast communication link. For big A computing time dominates schedule length, and it is profitable to distribute and parallelize computations. Hence $\frac{m'}{m}$ is growing. On the other hand, for big V the number of chunks must be big anyway, communication time (mainly startup times S_i) is dominating over computation time, and A is less important in determining schedule length. Therefore, A is not influencing m' for big V.

In Fig. 6c dependence of $\frac{m'}{m}$ on $\frac{B}{V}$ is shown. In our method of test instance generation average number of processors is ≈ 5. Hence, for $\frac{B}{V} < \frac{1}{5}$ the memory space necessary to process load V is created by using many load chunks, and many processors working in parallel. On the other hand, when $\frac{B}{V} > \frac{1}{5}$ the size of memory is sufficient to process the whole load in just one installment. Therefore, good solutions tend to use only a few processors with fast communication and computation.

Fig. 6d shows relation between S, V, and $\frac{m'}{m}$. With growing amount of load V the number of different used processors is increasing as in other experiments. For small V the number of different used processors decreases with growing S which

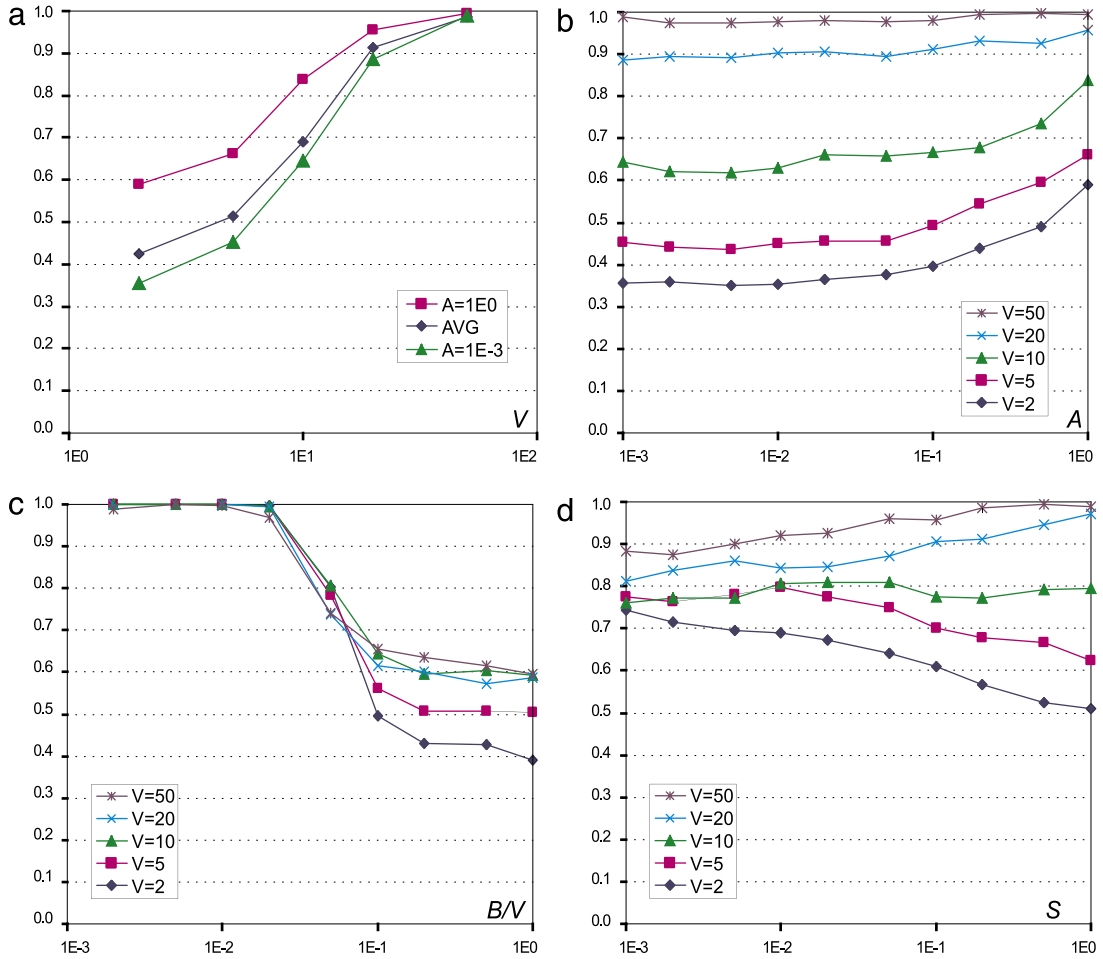

Fig. 6. Relative number $\frac{m'}{m}$ of different used processors in the solutions of GA (a) vs. V, (b) vs. A (c) vs. $\frac{B}{V}$, (d) vs. S.

is in accord with our earlier expectations. However, for big V the increasing S results in increased m'. This counterintuitive behavior can be partially explained by the way of generating test instances. Note that startup times of all processors are equal in the experiments depicted in Fig. 6d. When V is big then the number of sent chunks must be also big. With growing S startup times dominate in the schedule length and other parameters, by which the processors differ, are becoming meaningless. Therefore, GA becomes myopic to the differences in processor parameters, and hence more processors are randomly drawn to the solutions.

Dependence of $\frac{m'}{m}$ on C (not shown here) is very weak. This is again a very surprising situation because in many DLT papers communication rate C was considered crucial for system performance. Only for small V and big C (close to 1) is the number of used processors slightly decreasing with growing C. This is a result of the startup time domination in communication time. Only for small V the number of messages is small and hence startups time is small. Then, GA optimizes the schedule by using a few fast processors. This result does not eliminate C as an important performance determinant, as will be shown in the following study.

We finish this section with the following conclusions. The number of different used processors differs depending on the settings. In general it is increasing with V. In our experiment setting startup times dominated schedule length, especially when the number of chunks had to be big because V was big or B was small. When A is big and computation time is at least comparable with the communication time, then it is profitable to use many processors to parallelize computations. When C is big and its contribution to the communication time is comparable, or greater, than the contribution of the startup times, then it is profitable to choose few processors with small C.

4.4. Dominating set of processors

In the previous section we considered the number of different used processors, not the degree of participation in computations. Here we analyze distribution of the load between the processors. We want to determine if there is any inequality in the load distribution, and if it is the case, then what kind of processors dominate in the computations.

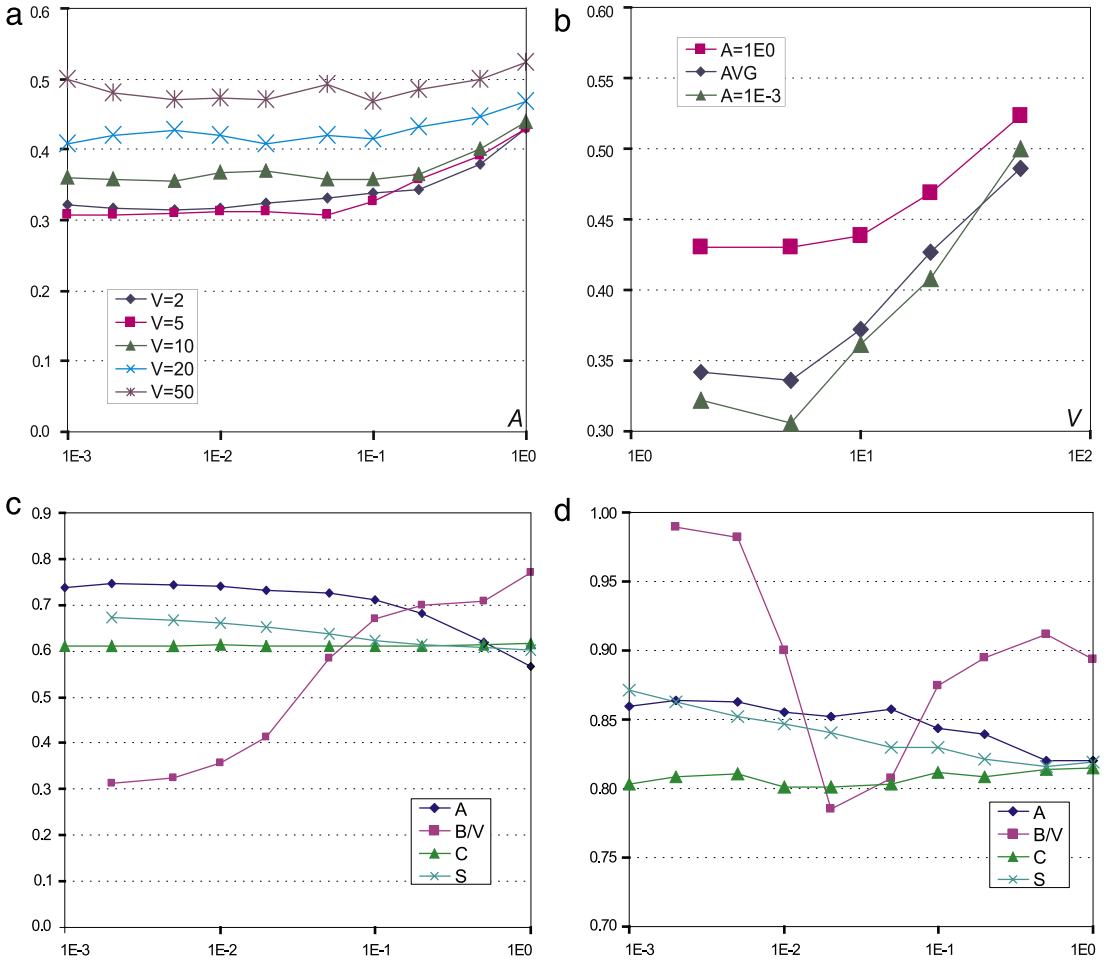

Fig. 7. Load frequent processor sets in GA solutions. (a) Fraction of frequent processors vs. A, (b) fraction of frequent processors vs. V, (c) fraction of load V on the most loaded processor vs. A, $\frac{B}{V}$, C, S, (d) fraction of load V on all the frequent processors vs. A, $\frac{B}{V}$, C, S.

A measure of processor domination in computations is based on the set of processors most frequently receiving the load or messages. Let $V_{\max}$ be the greatest total load received by any single processor. We will say that some set of processors is *load frequent* if it includes all processors which receive at least $\frac{V_{\max}}{2}$ units of load. The processors in load frequent set are called load frequent, or just frequent.

We want to examine how much load, and messages are sent to frequent processor sets. The results of this study are shown in Fig. 7. All values shown in this figure are relative. Thus, processor numbers are shown with respect to m, and the loads are shown relative to V. In Fig. 7c,d, the horizontal axes represent all parameters A, $\frac{B}{V}$, C, S, in range [0, 1] for four different relations. A general observation is that the functions of the number of load frequent processors in A (Fig. 7a), and in $\frac{B}{V}$, C, S (not shown here) have very similar tendencies as the functions of $\frac{m'}{m}$ in the above parameters (see Fig. 6). However, the range of changes of the frequent processor number vs. V is narrower than the range of changes in $\frac{m'}{m}$. For example, in Fig. 6a the number of used processors changes in range approx. [0.4, 1]. Here, the range of changes is approx. [0.3, 0.5] (Fig. 7b). Even smaller ranges were observed in the experiments with changing $\frac{B}{V}$, C, S. It can be concluded that the size of frequent set of processors is growing with V, but not as quickly as the number of different used processors m'. It is because only a selected set of processors is frequently used while many other processors get to the solution due to the randomized selection in GA.

In Fig. 7c the load of the most frequently used processor is depicted vs. changing A, $\frac{B}{V}$, C, S. Independently of the type of changes the most loaded processor receives 0.6–0.75V on average. With growing A computation time starts dominating in schedule length, the processor selection method tends to build more computing power, and more processors are appended to the frequent set. Hence, the greatest piece of load sent to a single processor is diminishing. Growing $\frac{B}{V}$ allows for using fewer processors and for economizing on communication time. Hence, for big $\frac{B}{V}$ the most loaded processor receives almost 0.75V. For small $\frac{B}{V}$ a big number of communications must be made anyway which expose the cost of communication startup times dominating in the schedule length. Consequently, GA becomes myopic to other processor parameters, the frequent

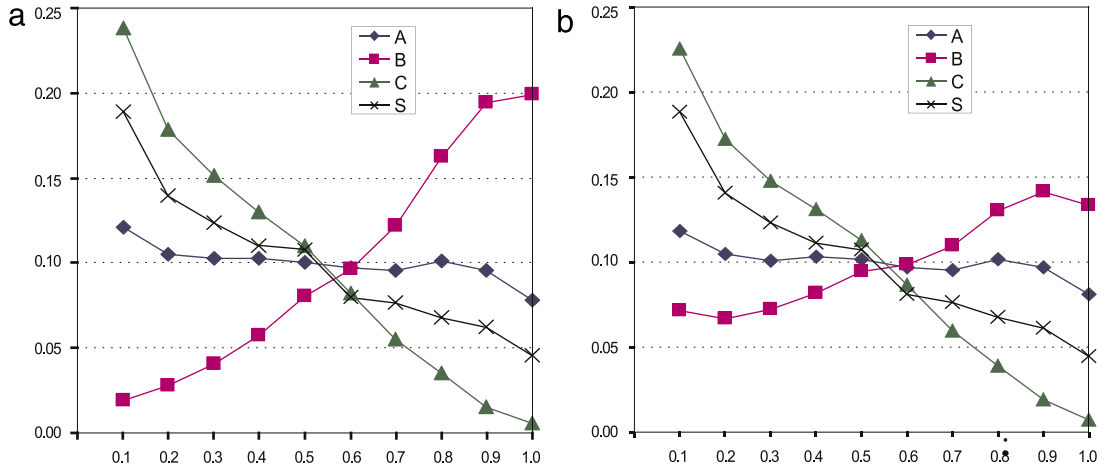

Fig. 8. Fraction of load and fraction of received messages vs. processor rank in GA solutions. (a) Fraction of load vs. rank, (b) Fraction of all chunks vs. rank.

set has more processors, and the load is more dispersed between the processors. The dependence on S shown in Fig. 7c is very weak. However, this is an average over many sizes V. A more detailed picture exposes diversity with V similar to the one shown in Fig. 6d, though in much narrower range. Unlike in Fig. 6d the load sizes are generally decreasing with S, even for small loads V. Similarly to the results in Section 4.3 the biggest piece of the load received by a single processor does not depend on C.

The sum of the load assigned to all frequent processors is shown in Fig. 7d. As it can be seen the frequent processor set collects more than $0.8V$ on average. The function of the total load vs. $\frac{B}{V}$ has a minimum. This unexpected phenomenon can be explained in the following way. For big values of $\frac{B}{V}$ only a few processors take part in the computation because a single installment is sufficient to process the whole load. Therefore, the number of messages is small, load chunks have sizes close to processor memory buffer sizes, the frequent set has small cardinality and receives whole load. With decreasing $\frac{B}{V}$ more and more processors receive some load, an the contribution of the most loaded processor(s) is decreasing as depicted in Fig. 7c. However, when $\frac{B}{V}$ becomes extremely small, communication cost is dominating schedule length, GA becomes unaware of processor parameters, and more of the processors are randomly included in the frequent set. Consequently, the cardinality of the frequent set is growing and the total load in the frequent set is growing.

Similar results were obtained in the analysis of the set of processors receiving the greatest number of messages (instead of the load). We finish the above exercise with a conclusion, that the frequent set of processors really exists. With the exception of the instances biased by small $\frac{B}{V}$ or big S, when almost all processors are frequent, the frequent set has approx. 40%–50% of all available processors. They received 80%–85% of the whole load, again with the exception of the cases biased by small $\frac{B}{V}$ or big S.

The above results confirm that the set of processors dominating in the computation exists. Yet, consider the method of test instance generation. When studying influence of a certain parameter, all processors have this parameter equal. We learned on the importance of the considered parameter via the consequences of its low, or high, values. But effects of the diversity of the given parameter were switched off. We did not verify how important this parameter could be if it had different values in the processor set. Therefore, a second set of 1000 instances were generated with $V = 100$, m generated from $U[1, 100]$, and $A, B, C, S \sim U[0, 1]$. We examined the fraction of the whole load and the number of received messages against the rank of the processor in certain parameter. The results of this study are shown in Fig. 8.

In Fig. 8 processors were grouped into sets comprising 10% of the processors ranked according to certain parameter. For example, value 0.3 on the horizontal axes in Fig. 8 represent processors with relative rank $\frac{i}{m}$ in the range (0.2, 0.3]. The four functions depicted in Fig. 8 correspond to four different rankings: according to A, B, C, S. Let us remind that for A, C, S smaller values represent better performance, and for B bigger values are better. The relationships are similar for the received load (Fig. 8a) and for the number of messages (Fig. 8b). Hence we will discuss only the load distribution. The distribution of the load is tightly connected with all processor parameters. It is evident that processors which have best communication links with respect to C and S, and the biggest memory buffers receive most of the load to process. The processors with small B, and big S, C, receive almost no load. For parameter A the relationship is weaker but it is still noticeable (coefficient of correlation between A and the upper limit of rank box interval is ≈ -0.84).

We finish the study of the dominating set of processors with the following observations made on the basis of computational experiments:

- the dominating processor set exists,
- the frequent processors set, as we defined it, comprises approximately 40%–50% of all processors,
- in the biased case of big S, and small $\frac{B}{V}$ frequent processor set may include nearly all processors,
- there is a strong correlation between parameters C_i, S_i, B_i, and the amount of load received for processing.

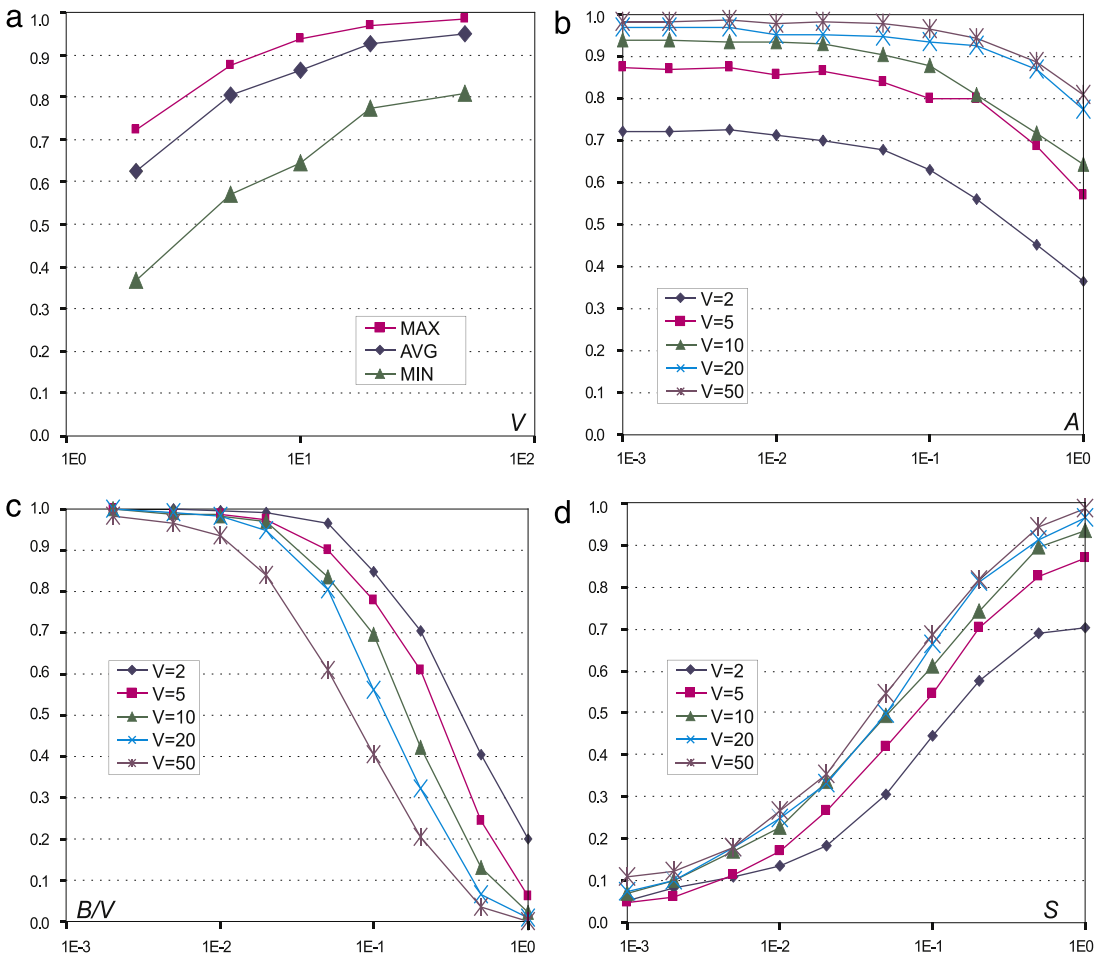

Fig. 9. Average number of full chunks in GA solutions (a) vs. V in experiments with changing A, (b) vs. A, (c) vs. $\frac{B}{V}$, (d) vs. S.

4.5. Chunk size saturation

Another feature of problem solutions is load partitioning. After determining the sequence of communication and the overlaps, a linear program was used to find load distribution. Since the computational cost of linear programming is high, it would be profitable to eliminate it in constructing good quality solutions. To examine the structure of load partitioning we analyzed the number of chunks whose sizes equal the size of the target processor buffer, i.e. $\alpha_i = B_{\sigma(i)}$. We will call such chunks *full chunks*. It would be attractive to use the processor buffer size as chunk size, thus eliminating the need for linear programming. The results of this exercise are shown in Fig. 9.

In all figures shown in Fig. 9 the number of full chunks is shown in relation to the total number of chunks n. The number of full chunks is almost always high or noticeable, but not all chunks are full. As it can be seen if Fig. 9a,b,d, with growing size V the number of full chunks is also growing. This is intuitively reasonable because bigger load V requires more messages which expose costs of the startup times S_i. These can be reduced by using as few messages as possible, and consequently filling the bufferers more completely. This is also confirmed in Fig. 9c where the number of full chunks is shown against changing $\frac{B}{V}$, and various values of V. When $\frac{B}{V}$ is small, the number of messages must be big, hence the startup times dominate in the schedule length, and to reduce their contribution, the buffers are more fully filled. This situation is repeated in Fig. 9d where the number of full chunks increases with the startup times. With growing A (Fig. 9b) the number of full chunks is decreasing because the computation time starts dominating in the schedule length, not the startup times. Note that in Fig. 9c the number of full chunks decreases with V, which may be attributed to the randomized nature of *GA*. With growing V greater number of messages must be sent. The message target processors are generated randomly. Hence with growing V chances are growing that a slow processor, or a processor with slow communication link may be selected to the communication sequence. The linear programming part of *GA* tries to minimize influence of such bad choices by reducing chunk sizes. Hence with growing V the fraction of chunks which are not full also grows.

4.6. When it is hard to find a good solution

In this section we analyze what makes our problem easy or hard to solve. Let us introduce the goal of this section in more detail. Heuristics build good quality solutions for many combinatorial optimization problems. However, this good performance often should be attributed to the nature of the problem, not a heuristic. Thus, it is possible that our genetic algorithm builds good solutions not because it is well designed, but because our scheduling problem may be easy to solve. If we learn which instances are easy, or hard, to solve then we will gain some new insights into the nature of the problem, and real merits of GA.

Now the problem is how to verify which instances are easy, and which ones are hard to solve. We will compare quality of the solutions obtained in three ways for various types of instances. The worst solution observed indicates how bad a solution may be. The random solutions are not biased to being good or bad. GA solutions represent solutions which are optimized, and supposed to be good. The three solution types indicate what can be expected in the worst case, achieved without great efforts (random solutions), and at considerable cost of optimization (GA). The first two types of solutions demonstrate the nature of the scheduling problem. Comparing the third type of solution with the first two shows efficiency of our optimization attempts. If GA solutions did not differ much from the random solutions, then it would signify bad GA design. All three algorithms were obtained using the GA infrastructure. The random solution is the best one in the initial GA population of $G = 20$ solutions. The worst solution is the worst one observed in the course of solving some instance by GA. These are solutions of combinatorial part of our problem, i.e. communication sequence σ, and overlaps δ_{ij}. In all three algorithms linear programming was used to obtain the best chunk sizes α_i, and schedule length, for the given combinatorial part of the solution. Quality of the solutions is measured as the relative distance from the lower bound calculated in the following way. The minimum communication time is $\tau_1 = n_{\mathrm{MIN}} S_{\min} + V C_{\min}$, where $C_{\min} = \min_{i=1}^{m}\{C_i\}$, $S_{\min} = \min_{i=1}^{m}\{S_i\}$. In time τ_1 at most $V_0 = (\tau_1 - S_{\min}) \sum_{i=1}^{m} \frac{1}{A_i}$ load could be processed. The remaining load $V - V_0$ is processed in time at least equal to $\max\{0, \frac{V}{\sum_{i=1}^{m} 1/A_i} - \tau_1 + S_{\min}\}$. Thus the lower bound is equal to $\tau_1 + \max\{0, \frac{V}{\sum_{i=1}^{m} 1/A_i} - \tau_1 + S_{\min}\}$. We will examine performance of the above three types of solutions for changing values of system parameters and their dispersion.

In Fig. 10 we have shown influence of the system parameters on the quality of the above three solution types. Fig. 10a,b,c show results for the first set of random instances and $V = 20$. It is striking that the worst case solutions (denoted *WRST*) can be over one order of magnitude further from the lower bound than the random solutions (denoted *RND*) or the solutions of the genetic algorithm (denoted *GA*). Moreover, *GA* solutions are substantially better than *RND* solutions. Hence, *GA* really works. Now let us analyze the tendencies in Fig. 10a,b,c. As it can be seen in Fig. 10a with growing C all the lines tend to 1 which means that as communication speed is decreasing, schedule length becomes dominated by the time of sending load off the originator. Hence in such a biased case it is getting easier to obtain good solutions. Similar tendency was observed for growing parameter A (not shown here).

In Fig. 10b the dependence of solution quality on changing $\frac{B}{V}$ is shown. With growing $\frac{B}{V}$ all the three types of solutions get closer to the lower bound. It is intuitively attractive to conclude that with growing $\frac{B}{V}$ good solutions are easier to obtain because we are less limited with the choice of the processor. Not disregarding this growing flexibility, it should not be forgotten that the construction of the lower bound influences the results presented here. The lower bound is based on the assumption that the smallest S_i coincide with the biggest B_i, which is rarely true. Hence, for small $\frac{B}{V}$ and big number of the startups the error resulting from this simplification is significant. With increasing $\frac{B}{V}$ the domination of the startup costs in the schedule length decreases, and the lower bound is representing this situation better. Thus, the results in Fig. 10b indeed confirm that with growing $\frac{B}{V}$ it is getting easier to obtain good quality solutions, however, it is achieved by using fewer messages and communication startup times. Moreover, for the biggest $\frac{B}{V}$ solutions *WRST*, *RND* are getting slightly worse and *GA* solutions are not. This means that even if memory buffers are big, it its necessary to adjust the set of used processors. Genetic algorithm is doing it better than in the *RND* solutions.

In Fig. 10c dependence of the three types of solutions on changing parameter S is shown. A counterintuitive tendency of improving *WRST* solution quality with growing S can be observed. With growing S the contribution of the startup time to schedule length is growing, independently of the chosen set of processors. Therefore, the difference between the worst solution and the lower bound is decreasing with growing S. Genetic algorithm is performing better than *RND* because it is able to build solutions with relative quality improving even with increasing domination of the startup time.

In Fig. 10d quality of the solutions for growing dispersion of S is shown. The test instances for Fig. 10d were generated as in the first set of instances with $V = 20$, except for parameter S which was generated with uniform distribution from range $[\frac{1-\delta_S}{2}, \frac{1+\delta_S}{2}]$. The value of δ_S is shown on the horizontal line in Fig. 10d. As it can be seen, with growing δ_S, and hence with growing heterogeneity of the system, quality of all three types of solutions is worsening. This means that our problem is becoming harder to solve with growing heterogeneity of the computing environment. Similar experiments were performed for controlled dispersion δ_A, δ_B, δ_C of parameters A, $\frac{B}{V}$, C, respectively. In all cases the dependence of the quality of solutions on the range of diversity has a very similar shape which once again confirms that by nature of our scheduling problem in heterogeneous systems good quality solutions are harder to obtain. Let us use the range of change of the worst-case solutions quality as an indicator of the sensitivity to the dispersion of certain parameter. For δ_S changing from 1E-3 to 1 the average distance from the lower bound grew $\approx$34 times. For similar changes of: (1) δ_C the distance changed $\approx$14 times,

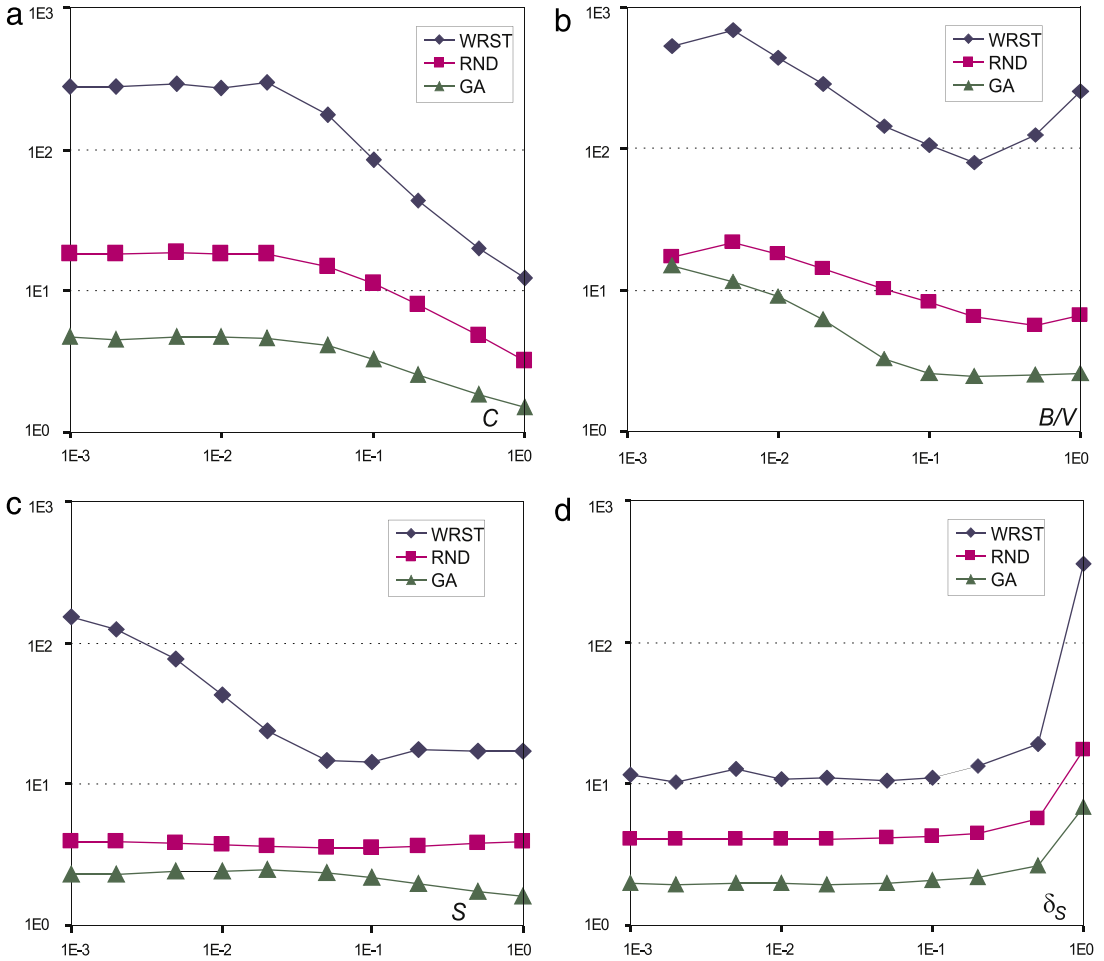

Fig. 10. Quality of the solutions with reference to the lower bound for $V = 20$ (a) vs. C, (b) vs. $\frac{B}{V}$, (c) vs. S, (d) vs. the dispersion of S.

(2) δ_A changed $\approx$1.8 times, (3) δ_B changed $\approx$1.3 times. This means that diversity of S, C have the strongest influence on difficulty of obtaining good solutions, and diversity of A, $\frac{B}{V}$ the smallest.

We finish this section with the following conclusions.

- It follows from the nature of our scheduling problem that it is easier to obtain high quality solutions when communication time or computation time dominates in the schedule length.
- It is easier to obtain good solutions for big memory buffers.
- It is easier to obtain good solution for homogeneous systems. Solution quality is particularly sensitive to the dispersion of communication parameters S, C, and less to the dispersion of A, $\frac{B}{V}$.
- Genetic algorithm really works, because it builds considerably better solutions than *RND*. Moreover, in some cases it is able to counteract the general tendencies of solution quality represented in *RND*, *WRST*.

5. Conclusions

In this paper we analyzed scheduling divisible loads on heterogeneous systems with communication startup times, limited memory buffers and multi-round load distribution. A new, more realistic model of memory management was assumed. The scheduling problem has dual nature: combinatorial and algebraic. Two methods for solving the combinatorial part were proposed: branch-and-bound algorithm, and a genetic algorithm. The algebraic part is solvable by a linear program on condition that a solution from the combinatorial part is provided. The branch-and-bound algorithm turned out to be impracticable due to its prohibitive complexity. Therefore, in the following studies we relied on the solutions from the genetic algorithm.

In the second part of the paper we studied features of the solutions of our scheduling problem. The study was performed both analytically and by extensive computational experiments. The following observations were made:

- It has been established [12] that in the worst case arbitrarily big number of chunks may have to be accumulated in the optimum solutions. By this feature multiprocessor schedules differ from the optimum schedules on a single processor. However, it turned out that in the near-optimum solutions obtained by the genetic algorithm accumulating the chunks is very rare.
- There is a minimum number of messages that must be sent anyway. It can be shown that using this number of communications may result in arbitrarily bad solutions [12]. In computational experiments it has been established that the number of messages is a small multiple of the minimum possible (1.4-10). Communication startup time is the main disincentive to using great number of messages in delivering the load to the processors.
- There is inequality in load distribution and a dominating set of processors receives most of the load. The size of the dominating set of processors is growing with size of the load V. There is a strong correlation between the parameters of a processor, and its contribution in load processing. Processors with faster communication links, bigger memory buffers, and computing faster receive more load. It appears that the order of parameter importance in load distribution is C_i, B_i, S_i, A_i.
- Majority of load chunks, though not all, carry maximum load, i.e. equal to the size of processor buffer. The number of full chunks grows with V, and is strongly correlated with S_i, B_i.
- The problem has natural tendency to become easier to solve when one parameter dominates in the schedule length. For example, big values of all A_i, in relation to C_i, S_i simplify obtaining good solutions.
- Another side of the above observation is that it is relatively easy to build biased instances whose solutions are dictated by extreme values of certain parameter, e.g. extremely slow communication, or computation, or very small memory buffers.
- In a sense, parameters B_i and S_i go together when building a biased instance. Small memory buffers B_i incur many communications which expose cost of the startup time S_i. And vice versa, big startup times may be compensated by use of long messages which require big memory buffers.
- Good quality solutions are harder to obtain in heterogeneous systems.

We believe that the above set of observations may be helpful in constructing new, faster and yet effective heuristics for the above scheduling problem.

Acknowledgement

The research was partially supported by Polish Ministry of Science and Higher Education grant number NN 519 1889 33.

References

[1] R. Agrawal, H.V. Jagadish, Partitioning techniques for large-grained parallelism, IEEE Transactions on Computers 37 (1988) 1627–1634.
[2] Y.-C. Cheng, T.G. Robertazzi, Distributed computation with communication delay, IEEE Transactions on Aerospace and Electronic Systems 24 (1988) 700–712.
[3] V. Bharadwaj, D. Ghose, V. Mani, T. Robertazzi, Scheduling Divisible Loads in Parallel and Distributed Systems, IEEE Computer Society Press, Los Alamitos CA, 1996.
[4] M. Drozdowski, Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems, in: Series: Monographs, vol. 321, Poznań University of Technology Press, 1997, http://www.cs.put.poznan.pl/mdrozdowski/txt/h.ps.
[5] T. Robertazzi, Ten reasons to use divisible load theory, IEEE Computer 36 (2003) 63–68.
[6] X. Li, V. Bharadwaj, C.C. Ko, Processing divisible loads on single-level tree networks with buffer constraints, IEEE Transactions on Aerospace and Electronic Systems 36 (2000) 1298–1308.
[7] M. Drozdowski, P. Wolniewicz, Divisible load scheduling in systems with limited memory, Cluster Computing 6 (2003) 19–29.
[8] M. Drozdowski, P. Wolniewicz, Optimum divisible load scheduling on heterogeneous stars with limited memory, European Journal of Operational Research 172 (2006) 545–559.
[9] M. Drozdowski, M. Lawenda, Multi-installment divisible load processing in heterogeneous systems with limited memory, in: R. Wyrzykowski, et al. (Eds.), PPAM 2005, in: Lecture Notes in Computer Science, vol. 3911, Springer, Berlin, 2006, pp. 847–854.
[10] M. Drozdowski, M. Lawenda, A new model of multi-installment divisible loads processing in systems with limited memory, in: R. Wyrzykowski, et al. (Eds.), PPAM 2007, in: Lecture Notes in Computer Science, vol. 4967, Springer, Berlin, 2008, pp. 1009–1018.
[11] Lp_solve reference guide, 2007. http://lpsolve.sourceforge.net/5.5/.
[12] J. Berlińska, M. Drozdowski, M. Lawenda, Multi-Installment Divisible Loads Scheduling in Systems with Limited Memory, Research Report RA-07/08, Institute of Computing Science, Poznan University of Technology, 2008 http://www.cs.put.poznan.pl/mdrozdowski/raplIn/ra0708.pdf.
[13] V. Bharadwaj, D. Ghose, V. Mani, Multi-installment load distribution in tree networks with delays, IEEE Transactions on Aerospace and Electronic Systems 31 (1995) 555–567.
[14] Y. Yang, H. Casanova, M. Drozdowski, M. Lawenda, A. Legrand, On the complexity of multi-round divisible load scheduling, Research Report 6096, INRIA Rhône-Alpes, 38334 Montbonnot Saint Ismier, France, January 2007. https://hal.inria.fr/inria-00123711.